# Recursive Types: the syntactic and semantic approaches(Type Theory and its Applications to Computer Systems)

AUTHOR(S):

Coppo, Mario

数理解析研究所講究録
1023巻 1998年 16-41



# Recursive Types: the syntactic and semantic approaches

Mario Coppo

Università di Torino
Dipartimento di Informatica
Corso Svizzera 185
10149 Torino (Italy)
e-mail: coppo@di.unito.it

**Abstract.** The aim of this paper is to study some basic syntactic properties of type inference systems with recursive types, and in particular normalization, subject reduction and the existence of principal type schemes. We do not intend to present original results. Most of material presented here, except some results about the decidability of type inference, has already been subject of some other papers (the main sources are [18], [12] and [5]) or is part of the folklore of the subject. Our main contribution is to have put it in a uniform frame.

## 1 Introduction

One of the most interesting notions of type constraint for functional programming languages is the one derived from Curry's Functionality Theory ([7]), which has suggested the type disciplines incorporated now in most functional programming languages, notably ML ([16]) and Miranda([19]). Several features make this sort of polymorphism particularly attractive both from the practical and the theoretical point of view. The type inference algorithm is complete, due to the existence of principal type schemes ([8], [15]) which fully characterize the set of types assignable to each term. Moreover it has been proved that the inference system has good syntactic properties like the subject reduction theorem which implies that terms having type in this discipline cannot produce run time errors. Another remarkable property of this system is that typed terms have always a strong normal form.

In the present paper we study extensions of the basic type inference system for the $\lambda$-calculus to include recursive type definitions as a means of introducing new types. A usual way of introducing new types in programming languages is by means of equations in which the type symbol being defined occurs in the term defining it. For example, the type of lists of objects of type $A$ can be specified by the equation

$$A\text{-list} = nil + (A \times A\text{-list}) \tag{1}$$

assuming the existence of a type constant $nil$ for the one element type, and type constructors $+$ for disjoint union of types and $\times$ for cartesian product.

Even when the only type constructor available is the function type constructor $\rightarrow$, it is well known that it is nevertheless useful to have some means for defining circular type expressions. For example, assuming a type $c$ such that $c = c \rightarrow A$ (where $A$ is any type) one can assign type $(A \rightarrow A) \rightarrow A$ to the fixed point combinator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, permitting in this kind of system recursion over values without having to introduce it explicitly in the base language by means of a new constant. This means also that when considering recursive types, in general, the property of strong normalization is lost.

We will address inference system in which recursive types are defined by means of equations of the shape $B = C$, where $B$ and $C$ are simple type expressions. Following [4] we call *system of type constraints* a set of equations like this. The most interesting case is, obviously, when $B$ is an atomic type $c$ and $C$ is an arbitrary type expression possibly containing $c$. This amount to define type $c$ with the property of being equal to $C$.

When we assume a set $\mathcal{C}$ of type constraints we must consider a notion of equivalence between types induced by the equations in $\mathcal{C}$. There are however at least two ways of defining such equivalence. In one more restrictive and syntactic way (*weak* equivalence) we consider two type $A$ and $A'$ equivalent only if they con be obtained one from the other by replacing subtypes occurring in the l.h.s. some equations of $\mathcal{C}$ by the corresponding types occurring in r.h.s. of the same equation (or vice-versa). This amounts to consider the least congruence induced by the equations in $\mathcal{C}$. Weak equivalence turns out to be easily formalizable and to have nice syntactic properties, like subject reduction and the existence of a notion of principal type scheme. It is also well know ([13]) a characterization of the class or recursive type definitions which guarantee the strong normalization of the terms that can be typed from them.

There is however another notion of type equivalence (*strong* equivalence) which amounts to consider equivalent two types if they, seen as binary trees, can be made equal at any finite level by repeatedly applying the equations in $\mathcal{C}$. This notion of equivalence is stronger than the weak one and correspond, informally, to considering equivalent two types if they represent the same notion of behavior. This is, for instance, the notion of type equality determined by the type checking algorithms of the programming language ALGOL 60. Moreover strong equivalence is the notion of type equality induced by interpretations in continuous models (see [5]). We will show in this paper that strong equivalence has also good syntactic properties.

The aim of this paper is to study properties of type inference systems with recursive type definitions, and in particular normalization, subject reduction and the existence of principal type schemes. We do not intend to present original results. Most of material presented here, except some results about the decidability of type inference, has already been subject of some other papers (the main sources are [18], [12] and [5]) or is part of the folklore of the subject. Our main contribution is to have put it in a uniform frame.

# 2 Inference Systems with Recursive Types

## 2.1 Recursive Types and Type Equivalence

We will study the notion of recursive type starting from the somewhat more general notion of type algebra, which has been motivated by Scott [17] and formally developed in Breazu Tannen and Meyer [4]. In addition to being interesting by itself, this notion will also prove to be a useful technical tool later on.

Let $\mathbf{A}$ be a set of atomic types (we do not distinguish for the moment between variables and constants). $\mathsf{T}_{\mathbf{A}}$ denotes the set of types defined from the atomic types in $\mathbf{A}$ by the $\rightarrow$constructor.

**Definition 1.** Let $\mathbf{A}$ a set of atomic types. A *type algebra* is a pair $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ where $\simeq$ is a congruence over $\mathsf{T}_{\mathbf{A}}$ (i.e. is such that $A \simeq A'$ and $B \simeq B'$ implies $A \rightarrow A' \simeq B \rightarrow B'$).

We are mainly interested in type algebras where the congruence $\simeq$ can be generated by a finite set of type equations via equational reasoning. Let

$$\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$$

be a set of formal equations between types $A_i, B_i \in \mathsf{T}_{\mathbf{A}}$. Following [4] we call such a set of equations a *system of type constraints*.

As remarked in the introduction there are essentially two ways in which the equations of $\mathcal{C}$ can be extended to a congruence over $\mathsf{T}_{\mathbf{A}}$. We introduce them in the following subsections.

**Weak Equality** The simplest way to define a congruence from a system of type constraints $\mathcal{C}$ is to extended $\mathcal{C}$ to a congruence over $\mathsf{T}_{\mathbf{A}}$ by adding structural rules and transitivity. We do this defining formal systems in which we can prove judgments of the shape

$$\mathcal{C} \vdash A = B$$

whose meaning is that the type expression $A$ and $B$ can be proved equivalent from the equations in $\mathcal{C}$. This is done in the following definition where the rules for equational reasoning are given via a formal inference system $(\sim)$. We will call weak equivalence the notion of type equality so obtained.

**Definition 2.** Let $\mathcal{C} = \{A_i = B_i \mid 1 \leq i \leq n\}$, where $A_i, B_i \in \mathsf{T}_{\mathbf{A}}$, be a system of type constraints. The system $(\sim)$ is the system for equational reasoning defined

by the following axioms and rules.

$$\text{(eq)} \quad \mathcal{C} \vdash A = B \quad \text{if } (A = B \in \mathcal{C})$$

$$\text{(ident)} \quad \mathcal{C} \vdash A = A$$

$$\text{(symm)} \quad \frac{\mathcal{C} \vdash A = B}{\mathcal{C} \vdash B = A}$$

$$(\rightarrow) \quad \frac{\mathcal{C} \vdash A = A' \quad \mathcal{C} \vdash B = B'}{\mathcal{C} \vdash A \rightarrow B = A' \rightarrow B'}$$

$$\text{(trans)} \quad \frac{\mathcal{C} \vdash A = B \quad \mathcal{C} \vdash B = C}{\mathcal{C} \vdash A = C}$$

We also write $A \sim_{\mathcal{C}} B$ ($A$ is *weakly equivalent* to $B$ with respect to $\mathcal{C}$) to mean that $\mathcal{C} \vdash_{\sim} A = B$.

Note that $\sim_{\mathcal{C}}$ is the minimal congruence over $\mathsf{T}_{\mathbf{A}}$ generated by $\mathcal{C}$. We say also that $\mathcal{C}$ is a (finite) *presentation* of the type algebra $\langle \mathsf{T}_{\mathbf{A}}, \sim_{\mathcal{C}} \rangle$.
In informal notation, we will use $=$ to denote syntactic equality of types (modulo $\alpha$ conversion).

In the definition of recursive types we are mainly interested in the type algebras generated by systems of type constraints of the shape $c = C$ where $c \in \mathbf{A}$ is an atom and $C \in \mathsf{T}_{\mathbf{A}}$ is a non atomic type expression. This corresponds to the natural idea of defining type $c$ as equivalent to a type expression $C$ containing possibly $c$ itself. In this case we can consider $c$ as a new type constant.

**Definition 3.** A system of type constraints $\mathcal{R}$ is a *simultaneous recursion* (s.r.) if it has the form

$$\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}.$$

where, for all $1 \leq i \leq n$, $c_i \in \mathbf{A}$, $C_i$ is a non-atomic type expression over $\mathsf{T}_{\mathbf{A}}$ and $c_i \neq c_j$ for all $i \neq j$.

Note that equations like $c_i = c_i$ are forbidden in a s.r. and that we can dispose of any equation of the shape $c_i = c_j$ $(i \neq j)$ simply by replacing $c_i$ by $c_j$ in $\mathcal{R}$ (or vice versa).

We call *unfolding* the operation of replacing $c_i$ by $C_i$ in a type and *folding* the reverse operation. So, two types are weakly equivalent in $\mathcal{C}$ if they can be transformed one into the other by a finite number of applications of the operations of folding and unfolding.

*Example 1.* (i) The s.r.

$$\mathcal{R}_0 = \{c = c \rightarrow c\}$$

defines a type $c$ such that any pure $\lambda$-term $M$ has type $c$, by assigning type $c$ also to variables. In fact $c$ represents a type which satisfies the defining equation of models of the $\lambda$-calculus. Moreover we have $c =_{\mathcal{R}_0} c \rightarrow c$ and $c \rightarrow c \sim_{\mathcal{R}_0} c \rightarrow$

$(c \to c)$.
(ii) Let $\mathcal{R}_1 = \{c = A \to c\}$ where $A$ is any type. Let $T = A \to c$. Then we have

$$c \sim_{\mathcal{R}_1} A \to c \sim_{\mathcal{R}_1} A \to A \to c \sim_{\mathcal{R}_1} \ldots$$

Let now $\mathcal{R}_1' = \{c = A \to A \to c\}$. Notice that

$$c \sim_{\mathcal{R}_1} A \to A \to c \sim_{\mathcal{R}_1} A \to A \to A \to A \to c \sim_{\mathcal{R}_1} \ldots$$

Moreover If we define $T = A \to c$ and $T' = A \to A \to c$ e have $T \sim_{\mathcal{R}_1} T'$ but $T \not\sim_{\mathcal{R}_1'} T'$

*Remark.* In a s.r. $\mathcal{R}$, the atomic types $c_i$ can be seen as new types defined by the equations. In a type algebra which is presented by a generic system of type constraints (in which both sides can be non-atomic type expressions), this interpretation is not possible. In this case it is difficult to classify the atoms in **A** as constants or variables.

**Strong Equality** Consider Example 1 above. Simultaneous recursions like $\mathcal{R}_1$ and $\mathcal{R}_1'$, although not equivalent, seem to express the same informal behavior: that of a function which can be applied to an arbitrary number of objects yielding a function of the same type. $T$ and $T'$, indeed, converge eventually, using both $\mathcal{R}_1$ and $\mathcal{R}_1'$ to the same (infinite) type when $c$ is "pushed down" by repeated steps of unfolding. We can define another notion of equivalence between recursive type expressions regarding them as finitary descriptions of a special class of infinite trees: this approach is followed systematically in Cardone and Coppo [5] for an alternative representation of recursive types.

In order to introduce this class we need some basic notions about infinite trees, mostly drawn from Courcelle [6]. If $X$ is a set, $X^*$ denotes the set of all finite sequences of elements of $X$. The elements of $X^*$ are usually called the *words* over the *alphabet* $X$. As usual concatenation is represented simply by juxtaposition; $\epsilon$ denotes the empty word.

**Definition 4 Trees.** Let $[n]$ denote the set $\{1, \ldots, n\}$, and define a $\mathbf{A} \cup \{\to\}$-*tree* (or just *tree*, for short) as a partial function

$$\alpha : [2]^* \rightharpoonup \mathbf{A} \cup \{\to\}$$

satisfying the conditions:

if $uv \in \mathrm{dom}(\alpha)$, then also $u \in \mathrm{dom}(\alpha)$.
if $u2 \in \mathrm{dom}(\alpha)$, then also $u1 \in \mathrm{dom}(\alpha)$.
if $\alpha(u) = \to$ then $u1, u2 \in dom(\alpha)$.
if $\alpha(u) \in \mathbf{A}$ then $uv \notin dom(\alpha)$ for all $v \neq \epsilon$.

The set of trees will be denoted by $\mathrm{Tr}^\infty$. $\mathrm{Tr}^F$ is the set of *finite* trees over $\mathbf{A} \cup \{\to\}$ (we omit to explicitly mention **A** since it will be clear from the context.

Since $\mathsf{Tr}^F$ is clearly isomorphic to $\mathsf{T}$ (the set of simple types), we will often identify them, considering simple types as finite trees and vice versa. The operation of substitution can be easily extended to trees.

Among infinite trees, we single out trees having a certain periodic structure, allowing to look at them as solutions of (systems of) equations of the form

$$\xi = A[\xi],$$

where $A \in \mathsf{Tr}^F$. These are the *regular* trees.

**Definition 5.** (Regular trees)

(i) Given a tree $\alpha \in \mathsf{Tr}^\infty$ and a word $w \in \mathrm{dom}(\alpha)$, let $\alpha/w$ be the tree defined by the conditions:
- $\mathrm{dom}(\alpha/w) = \{u \in [2]^* : wu \in \mathrm{dom}(\alpha)\}$;
- $(\alpha/w)(u) = \alpha(wu)$, for all $u \in \mathrm{dom}(\alpha/w)$.

(ii) A tree is *regular* if the set $\{\alpha/w | w \in \mathrm{dom}(\alpha)\}$ is finite. The set of regular trees is denoted by $\mathsf{Tr}^R$.

□

It is easy to see that regular trees are closed under substitutions, i.e. if $\alpha, \beta_1, \ldots, \beta_n$ are regular trees then also $\alpha[t_1 := \beta_1, \ldots, t_n := \beta_n]$ is a regular tree, where $\{t_1, \ldots, t_n\}$ $(n \geq 0)$ are variables occurring in $\alpha$.

We can turn $\mathsf{Tr}^\infty$ into a metric space in the following way [6].

**Definition 6.** Let $v \in [2]^*$ and let $\alpha, \alpha' \in \mathsf{Tr}^\infty$. Let $\|w\|$ denote the length of $w$. Define

$$\mathrm{d}(\alpha, \alpha') = \begin{cases} 0 & \text{if } \alpha = \alpha' \\ 2^{-\delta(\alpha,\alpha')} & \text{if } \alpha \neq \alpha' \end{cases}$$

where $\delta(\alpha, \alpha')$ is the length of the minimum path $w$ such that $w \in \mathrm{dom}(\alpha)$, $w \in \mathrm{dom}(\alpha')$ and $\alpha(w) \neq \alpha'(w)$.

It is well known (Courcelle [6, §2.2]) that $\langle \mathsf{Tr}^\infty, \mathrm{d} \rangle$ is a complete metric space. Indeed, with respect to this topology, the set $\mathsf{Tr}^F$ is a dense subset of $\mathsf{Tr}^\infty$ and $\mathsf{Tr}^\infty$ is the topological completion of $\mathsf{Tr}^F$.

If $\langle D, d \rangle$ is a metric space a map $f : D \longrightarrow D$ is *contracting* if there exists a real number c $(0 \leq c < 1)$ such that

$$\forall x, x' \in D \;\; d(f(x), f(x')) \;\leq\; c \cdot d(x, x') \,.$$

A basic property of complete metric spaces is the following result:

**Theorem 7.** *Let $\langle D, d \rangle$ be a complete metric space. Every contracting mapping $f : D \longrightarrow D$ ha a* unique *fixed point in $D$ defined as the limit of the Cauchy sequence $\langle f^n(x_0)) \rangle_{n \geq 0}$, where $x_0$ is any element of $D$.* □

Now take a tree $\alpha \in \mathsf{Tr}^\infty$ and a variable $t$ which occurs in $\alpha$. Then, if $\alpha \not\equiv t$, $\lambda\zeta \in \mathsf{Tr}^\infty.\alpha[t := \zeta]$ defines a contracting mapping of $\mathsf{Tr}^\infty$ into itself. The following property is also easy to prove (Courcelle [6, Theorem 4.3.1]).

**Proposition 8.** *If* $\alpha \in \mathsf{Tr}^R$ *and* $\alpha \not\equiv t$ *then* $\mathbf{fix}(\lambda\zeta \in \mathsf{Tr}^\infty.\alpha[t := \zeta]) \in \mathsf{Tr}^R$.

A notion of equivalence between types defined via a s.r. can be obtained by considering two types equivalent if their corresponding trees, obtained by infinite unfolding using the equations in $\mathcal{C}$, are equal. We define an interpretation of types as infinite (regular) trees.

A *solution* in $\mathsf{Tr}^R$ of s.r. $\mathcal{R}$ is $n$-tuple $\langle \alpha_1, \ldots, \alpha_n \rangle \in (\mathsf{Tr}^R)^n$ such that $\alpha_i = C_i[c_1 := \alpha_1, \ldots, c_n := \alpha_n]$ for each $i$ such that $1 \leq i \leq n$.

**Theorem 9.** *A s.r.* $\mathcal{R}$ *has a unique solution in* $\mathsf{Tr}^R$. □

The existence and uniqueness of the solution follow from Banach fixed point theorem. In fact, a s.r. $\mathcal{R}$ induces a contracting mapping on the product space:

$$\lambda c_1 \ldots \lambda c_n.\langle C_1, \ldots, C_n \rangle : (\mathsf{Tr}^\infty)^n \longrightarrow (\mathsf{Tr}^\infty)^n$$

whose unique fixed point is a $n$-tuple $\langle \alpha_1, \ldots, \alpha_n \rangle$ is the solution of $\mathcal{R}$ in $\mathsf{Tr}^R$, as all its components are regular.

**Definition 10.** Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be a s.r.. If $\langle \alpha_1, \ldots, \alpha_n \rangle$ is the solution of $\mathcal{R}$ in $\mathsf{Tr}^R$, let $(\ -\ )^*_{\mathcal{C}} : \mathsf{T}_{\rightarrow} \mathsf{Tr}^R$ be the mapping defined by the following clauses:

- $(\phi)^*_{\mathcal{C}} = \phi$, for all $\phi \in \mathbf{A}$.
- $(A \rightarrow B)^*_{\mathcal{C}} = (A)^*_{\mathcal{C}} \rightarrow (B)^*_{\mathcal{C}}$.
- $(c_i)^*_{\mathcal{C}} = \alpha_i$, for $i = 1, \ldots, n$.

We can now define the tree equivalence induced by a s.r.:

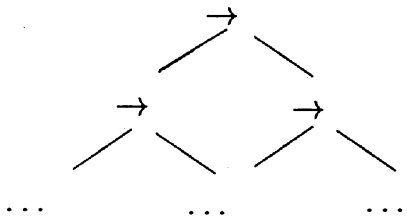
**Definition 11.** If $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ is a simultaneous recursion over $\mathsf{T}_{\mathbf{A}}$. The relation $\approx_{\mathcal{R}} \subseteq \mathsf{T}_{\mathbf{A}} \times \mathsf{T}_{\mathbf{A}}$ is defined by setting $A \approx_{\mathcal{R}} B$ if $(A)^*_{\mathcal{R}} = (B)^*_{\mathcal{R}}$. We say that $A$ is *strongly equivalent* to $B$ with respect to $\mathcal{R}$.

So, two types are strongly equivalent if they can be reduced to the same infinite tree by unfolding the unknowns occurring in them, by means of the equations of $\mathcal{R}$, infinitely many times. It is easy to see that $\approx_{\mathcal{R}}$ is a congruence Moreover note that $\approx_{\mathcal{R}}$ includes properly $\sim_{\mathcal{R}}$.

*Example 2.* Take the s.r. $\mathcal{C}_0$ defined in Example 1, and let $\mathcal{C}_2$ be the s.r. defined by the following equations:

$$c_1 = c_2 \rightarrow c_1$$
$$c_2 = c_1 \rightarrow c_2$$

It is easy to see that both $(c_1)^*_{\mathcal{C}_2}$ and $(c_2)^*_{\mathcal{C}_2}$ are equal to $(c)^*_{\mathcal{C}_0}$, therefore, in particular, $c_1 \approx_{\mathcal{C}_2} c_2$. Note that $c_1 \not\sim_{\mathcal{C}_2} c_2$. In fact, both $(c_1)^*_{\mathcal{C}_0}$ and $(c_2)^*_{\mathcal{C}_0}$ are equal to the infinite tree:



$\approx_{\mathcal{R}}$ which has clearly a more semantic nature as compared to $\sim_{\mathcal{R}}$ is the type equivalence induced by the interpretation of types in continuous models (see[5]). However, $\approx_{\mathcal{R}}$ has interesting syntactic properties, even if there are properties of terms (like strong normalization) that can be easily characterized with respect to $\sim_{\mathcal{R}}$ and for which there seems to be no a straightforward characterization in $\approx_{\mathcal{R}}$. In addition, the formal treatment of $\approx_{\mathcal{R}}$ requires sometimes more sophisticated techniques. We will see in the next section an axiomatization of $\approx_{\mathcal{R}}$which uses a kind of co-induction principle.

## 2.2 Type assignment

We are now ready to set up formal systems for assigning types to terms of the pure $\lambda$-calculus, following the approach called à la Curry in [10].

For the basic definitions about $\lambda$-calculus and type assignment systems we rely on [10]. The inference systems must contain a rule to handle type equivalence. We can indeed introduce a number of different systems according to which notion of type equivalence we consider.

We include the equations in $\mathcal{C}$, where $\mathcal{C}$ is a system of type constraints between the premises of the assignment, proving therefore judgments of the shape:

$$\mathcal{C}, \Gamma \vdash M : A,$$

which means that we can assign type $A$ to $M$ assuming the equations in $\mathcal{C}$ with respect to some notion of equivalence.

We define inference systems to assign types of the $\lambda$-calculus possibly containing constants, assuming for each constant $c$ a type $T_c \in \mathsf{T}_{\mathbf{A}}$.

**Definition 12.** Let $\mathcal{C}$ a system of type constraints. Let $\mathrm{R}_{\mathcal{C}}$ be one of $\sim$, $\approx$ (in the latter case we assume that $\mathcal{C}$ is a s.r.). Then the systems ($\lambda$R) are defined

by the following rules:

$$\text{(ax)} \qquad \mathcal{C}, \Gamma, x : A \vdash x : A$$

$$\text{(const)} \qquad \mathcal{C}, \Gamma \vdash c : T_c$$

$$(\to\text{-elim}) \quad \frac{\mathcal{C}, \Gamma \vdash M : A \to B \qquad \mathcal{C}, \Gamma \vdash N : A}{\mathcal{C}, \Gamma \vdash MN : B}$$

$$(\to\text{-intro}) \quad \frac{\mathcal{C}, \Gamma, x : A \vdash M : B}{\mathcal{C}, \Gamma \vdash \lambda x.M : A \to B}$$

$$\text{(equiv)} \quad \frac{\mathcal{C}, \Gamma \vdash M : A \qquad A \; \mathrm{R}_{\mathcal{C}} \; B}{\mathcal{C}, \Gamma \vdash M : B}$$

We will write $\mathcal{C}, \Gamma \vdash_{\lambda\sim} M : A$ ($\mathcal{C}, \Gamma \vdash_{\lambda\approx} M : A$) to denote deducibility in $(\lambda \sim)$ (or $(\lambda \approx)$). We show now some interesting typings for terms which have no type in the simple type assignment system without recursive types.

*Example 3.* (i) Let $A$ be any type. Take $\mathcal{R}_\Delta = \{c = c \to A\}$. The following deduction $\mathcal{D}_\Delta$ shows that $\mathcal{R}_\Delta \vdash_{\lambda\sim} \lambda x.xx : c$ in $(\lambda \sim)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{(ax)}}{\mathcal{R}_\Delta, \{x : c\} \vdash_{\lambda\sim} x : c} \qquad c \sim_{\mathcal{R}_\Delta} c \to A}{\mathcal{R}_\Delta, \{x : c\} \vdash_{\lambda\sim} x : c \to A}\,\text{(equiv)} \qquad \dfrac{\text{(ax)}}{\mathcal{R}_\Delta, \{x : c\} \vdash_{\lambda\sim} x : c}}{\mathcal{R}_\Delta, \{x : c\} \vdash_{\lambda\sim} (x\,x) : A}\,(\to\text{-elim})}{\mathcal{R}_\Delta, \vdash_{\lambda\sim} \lambda x.xx : c \to A}\,(\to\text{-intro}) \qquad c \to A \sim_{\mathcal{R}_\Delta} c}{\mathcal{R}_\Delta, \vdash_{\lambda\sim} \lambda x.xx : c}\,\text{(equiv)}$$

(ii) Using $\mathcal{D}_\Delta$ we have immediately

$$\dfrac{\dfrac{\mathcal{D}_\Delta \qquad c \sim_{\mathcal{R}_\Delta} c \to A}{\mathcal{R}_\Delta, \vdash_{\lambda\sim} \lambda x.xx : c \to A}\,\text{(equiv)} \qquad \mathcal{D}_\Delta}{\mathcal{R}_\Delta, \vdash_{\lambda\sim} ((\lambda x.xx)(\lambda x.xx)) : A}\,(\to\text{-elim})$$

(iii) Let $\mathcal{D}_{\Delta_f}$ denote a deduction of $\{f : A \to A\} \vdash_{\lambda\sim} \lambda x.f(x\,x) : c \to A$ in $(\lambda \sim)$ (it can be easily obtained modifying $\mathcal{D}_\Delta$). We can prove $\vdash_{\lambda\sim} Y : (A \to A) \to A$ in the following way.

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_{\Delta_f} \qquad c \to A \sim_{\mathcal{R}_\Delta} c}{\mathcal{R}_\Delta, \{f : A\} \vdash_{\lambda\sim} \lambda x.(f(x\,x)) : c}\,\text{(equiv)} \qquad \mathcal{D}_{\Delta_f}}{\mathcal{R}_\Delta, \{f : A\} \vdash_{\lambda\sim} (\lambda x.f(x\,x))(\lambda x.f(x\,x)) : A}\,(\to\text{-elim})}{\mathcal{R}_\Delta, \vdash_{\lambda\sim} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (A \to A) \to A}\,(\to\text{-intro})$$

The previous examples shows that the (strong) normalization theorem does not hold, in general, for recursive types.

The same statement can obviously be proved also in the stronger system $(\lambda \approx)$. Notice, however, that the two systems $(\lambda \sim)$ and $(\lambda \approx)$ are not equivalent but $(\lambda \sim)$ is weaker than $(\lambda \approx)$.

*Example 4.* Take $\mathcal{R}'_\Delta = \{c = (c \to A) \to A\}$. Then we have

$$\mathcal{R}'_\Delta, \{x : c\} \vdash_{\lambda\approx} x : c \to A$$

but $\mathcal{R}'_\Delta, \{x : c\} \not\vdash_{\lambda\sim} x : c \to A$

We can assume more generally that the notion of type equivalence is that associated to a generic type algebra $\langle \mathsf{T}_\mathbf{A}, \simeq \rangle$. Let us call $(\lambda \simeq)$ the system of type assignment in which type equivalence (in rule (equiv)) is $\simeq$. We denote $\Gamma \vdash_{\lambda\simeq} M : A$ if the statement $M : A$ con be proved from $\Gamma$ using $\simeq$ in rule (equiv). We will use such systems to prove some properties of type algebras in Section 4.

## 3 Properties of type equalities

In this section we study the properties of recursive types independently of their uses in typing $\lambda$-terms. In the next section we will connect them to the properties of type inference systems.

### 3.1 Invertibility

An important property of recursive types equivalences, is that of invertibility, which turns out to be crucial in the proof of the subject reduction theorem in section 4.

**Definition 13.** (Invertibility) We say that a type algebra $\langle \mathsf{T}_\mathbf{A}, \simeq \rangle$ is *invertible* if, for every $A, B, A', B' \in \mathsf{T}_\mathbf{A}$:

$$(A \to B) \simeq (A' \to B') \text{ imply } A \simeq A' \text{ and } B \simeq B'.$$

If an invertible congruence is generated by a set of type constraints $\mathcal{C}$ we say that $\mathcal{C}$ is invertible.

If $\mathcal{R}$ is a s.r. invertibility holds trivially for $\approx_\mathcal{R}$. This follows immediately from the definition of the mapping $(\ -\ )^*$ (see Def. 11). The proof that weak equivalence has the invertibility property requires a little more effort.

Invertibility for recursive types defined by a s.r. $\mathcal{R}$ can be proved via the construction of a Church-Rosser and strongly normalizing term rewriting system which generates $\sim_\mathcal{R}$. A immediate corollary of this proof is also the decidability of weak equality. The proof given here is due to Statman [18]. A more algebraic but less direct approach is given in Marz [12].

Let $\mathcal{C}$ and $\mathcal{C}'$ be two system of type constraints over the same set $\mathsf{T}_\mathbf{A}$ of types. $\mathcal{C} \vdash_\sim \mathcal{C}'$ means that $\mathcal{C} \vdash_\sim A{=}B$ for all equation $A{=}B \in \mathcal{C}'$. $\mathcal{C}$ and $\mathcal{C}'$ are *equivalent* (notation $\mathcal{C} \sim_\mu \mathcal{C}'$) if $\mathcal{C} \vdash_\sim \mathcal{C}'$ and $\mathcal{C}' \vdash_\sim \mathcal{C}$. The following definition is taken from Statman [18]:

**Definition 14.** A system of type constraints $\mathcal{R}'$ is an *expansion* of a s.r. $\mathcal{R}$ if $\mathcal{R}' = \mathcal{R} \cup \mathcal{E}$, where $\mathcal{E}$ is a set of equations of the form $a = c$, where $a$ is an atomic type not occurring in $\mathcal{R}$ and $c$ is an unknown of $\mathcal{R}$, such that $a = c, a = c' \in \mathcal{E}$ implies that $a{=}a'$. We also say that $\mathcal{R}'$ is an expanded s.r..

Note that $\mathcal{R}'$ is is just a trivial extension of $\mathcal{R}$ from a logical point of view and has, consequently, the same properties.

Given now a s.r. $\mathcal{R}$ we define a term rewriting system obtained by orienting the equations of $\mathcal{R}$.

**Definition 15.** Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be a s.r. The rewriting system $\mathrm{Trs}(\mathcal{R})$ consists of all rewriting rules $C_i \rightsquigarrow c_i$ for all $c_i = C_i \in \mathcal{R}$.

Note that $\sim_\mathcal{R}$ is the convertibility relation over $\mathsf{T}_\mathbf{A} \times \mathsf{T}_\mathbf{A}$ generated by $\mathrm{Trs}(\mathcal{R})$.

It is easy to see that $\mathrm{Trs}(\mathcal{R})$ is strongly normalizing, because each contraction decreases the size of the type to which it is applied. However, it is not, in general, Church-Rosser, as we show in Example 5 below. We can however transform the given s.r. into an equivalent one which is indeed Church-Rosser, via a construction which amounts to Knuth-Bendix completion (see [11]).

*Example 5.* Let $\mathcal{R}$ be the system

$$\begin{aligned} c_0 &= c_0 \to c_2 \\ c_1 &= (c_0 \to c_2) \to c_2 \\ c_2 &= c_0 \to c_1 \end{aligned}$$

Then $\mathrm{Trs}(\mathcal{R})$ consists of the rules

$$\begin{aligned} c_0 \to c_2 &\rightsquigarrow c_0 \\ (c_0 \to c_2) \to c_2 &\rightsquigarrow c_1 \\ c_0 \to c_1 &\rightsquigarrow c_2. \end{aligned}$$

Observe that the l.h.s. of the first equation is a subterm of the l.h.s. of the second one. In particular $(c_0 \to c_2) \to c_2$ can be reduced both to $c_1$ and to $c_0 \to c_2$ which further reduces to $c_0$: it has then two distinct normal forms $c_1$ and $c_0$. Therefore $\mathrm{Trs}(\mathcal{R})$ is not confluent (i.e., does not have the Church-Rosser property).

Expressions like $c_1$ and $c_0 \to c_2$ in the example above are called *critical pairs* in the literature on term rewriting systems. In $\mathrm{Trs}(\mathcal{R})$ there is a critical pair whenever there are $i, j$ such that $i \neq j$ and $C_i$ is a subexpression of $C_j$. By a

well known result of Knuth and Bendix (see [11, Corollary 2.4.14]) a strongly normalizing term rewriting system without critical pairs is Church-Rosser.

We give now, following ideas of Statman ([18]) an algorithm for transforming any s.r. into an expanded s.r. without critical pairs which has, therefore, the Church-Rosser property.

Given a s.r. $\mathcal{R}$ we define two sequences of sets of equations $\mathcal{D}_n$, $\mathcal{E}_n$, where $\mathcal{D}_n$ is a s.r. and $\mathcal{E}_n$ is an expansion of it. $\mathcal{D}_n$ and $\mathcal{E}_n$ are defined in such a way that, for all $n$, $\mathcal{D}_n \cup \mathcal{E}_n$ is an expansion of $\mathcal{D}_n$ equivalent to $\mathcal{R}$.

In the following definition we assume, without loss of generality, that there is a total order $<$ on the set $\mathrm{Unk}(\mathcal{R})$.

**Definition 16.** (Completion of $\mathcal{R}$) Let $\mathcal{R}$ be a s.r. and $<$ be a total order on $\mathrm{Unk}(\mathcal{R})$. We define by induction on $n$ a sequence of sets of equations $\mathcal{D}_n$, $\mathcal{E}_n$ $(n \geq 0$.
Let $\mathcal{D}_0 = \mathcal{R}$ and $\mathcal{E}_0 = \emptyset$.
Define $\mathcal{D}_{n+1}$, $\mathcal{E}_{n+1}$ from $\mathcal{D}_n$, $\mathcal{E}_n$ $(n \geq 0)$ in the following way:

1. if there exists a pair of equations $c_i = C_i, c_j = C_j \in \mathcal{D}_n$ such that $C_j$ is a proper subexpression of $C_i$ take
$$\mathcal{D}_{n+1} = (\mathcal{D}_n - \{c_i = C_i\}) \cup \{c_i = C_i^*\}$$
$$\mathcal{E}_{n+1} = \mathcal{E}_n.$$
where $C_i^*$ is the result of replacing all occurrences of $C_j$ in $C_i$ by $c_j$.
2. If there exist two equations $c_i = C, c_j = C \in \mathcal{D}_n$ then, assuming $c_i < c_j$, take
$$\mathcal{D}_{n+1} = \mathcal{D}_n[c_j := c_i]$$
$$\mathcal{E}_{n+1} = \mathcal{E}_n \cup \{c_j = c_i\}.$$
3. otherwise take $\mathcal{D}_{n+1} = \mathcal{D}_n$ and $\mathcal{E}_{n+1} = \mathcal{E}_n$

In the above definition note that in $\mathcal{D}_n$ there can be several pairs of equations satisfying 1 or 2, so the sequence of $\mathcal{D}_n$, $\mathcal{E}_n$ is not uniquely determined. However any choice is equivalent for our purpose.

The following Lemma can be proved by a straightforward induction on $n$.

**Lemma 17.** *For all $n \geq 0$ $\mathcal{D}_n \cup \mathcal{E}_n$ is equivalent to $\mathcal{R}$.* □

Note that, for all $n$, $\mathcal{D}_n$ is a s.r. and $\mathcal{R}_n \cup \mathcal{E}_n$ is an expansion of it. This construction can be applied indeed to any expanded s.r. as well. So all the results that we prove in this section holds in general for all expanded s.r..

Let $N$ be the least $n$ such that $\mathcal{D}_{n+1} = \mathcal{D}_n$ and $\mathcal{E}_{n+1} = \mathcal{E}_n$. This value must certainly exist since, in both steps 1 and 2 of Definition 16, the total number of symbols in $\mathcal{D}_n$ strictly decreases. So we must eventually reach a value of $n$ for which neither 1 nor 2 apply.

**Definition 18.** Let $\mathcal{R}$ be a s.r.. Then the term rewriting system $\mathrm{Trs}^+(\mathcal{R})$ is defined by:
$$\mathrm{Trs}^+(\mathcal{R}) = \mathrm{Trs}(\mathcal{D}_N) \cup \{c_j \rightsquigarrow c_i \mid c_j = c_i \in \mathcal{E}_N\}$$

**Lemma 19.** *Let $\mathcal{R}$ be a s.r.. Then $Trs^+(\mathcal{R})$ is strongly normalizing and Church-Rosser.*

*Proof.* First note that $\mathrm{Trs}^+(\mathcal{R})$ is strongly normalizing since each reduction of a rule belonging to $\mathrm{Trs}(\mathcal{D}_N)$ reduces the size of the expression to which it is applied and each rule of the shape $c_j \rightsquigarrow c_i$ is such that $c_i < c_j$ and so no infinite sequence of such reductions is possible.

Now note that $\mathrm{Trs}^+(\mathcal{R})$ has no critical pairs. In fact $\mathrm{Trs}(\mathcal{D}_N)$ has no such pairs, otherwise we could apply step 1 or 2 of Definition 16 to $\mathcal{D}_N$. Moreover if $c_j = c_i \in \mathcal{E}_N$ then $c_j$ does not occur in $\mathcal{D}_N$ and there is no other equation of the form $c_j = c_{i'}$ in $\mathcal{E}_N$. In fact, if $c_j = c_i$ has been put in $\mathcal{E}_k$ at step 2 of Definition 16, for some $(0 < k \leq N)$, then $c_j$ does not occur in $\mathcal{D}_k$ and, then, in $\mathcal{D}_n$ for all $n \geq k$. Consequently no other equation containing $c_j$ can be put in any $\mathcal{E}_n$ for $n > k$.
By the Knuth-Bendix theorem then $\rightsquigarrow$ is Church-Rosser. □

*Example 6.* Applying the above algorithm to the system $\mathcal{R}$ defined in Example 5 and assuming $c_0 < c_1 < c_2$ we have

$$\mathcal{D}_1 = \{c_0 = c_0 \to c_2,\ c_1 = c_0 \to c_2,\ c_2 = c_0 \to c_1\}$$
$$\mathcal{E}_1 = \emptyset$$

end

$$\mathcal{D}_2 = \{c_0 = c_0 \to c_2,\ c_2 = c_0 \to c_0\}$$
$$\mathcal{E}_2 = \{c_1 = c_0\}$$

no more transformations are possible, so we have $N = 2$. Note that $\mathcal{D}_2 \cup \mathcal{E}_2$ is equivalent to $\mathcal{R}$ and has no critical pairs.

Since $\mathcal{D}_N \cup \mathcal{E}_N$ is equivalent to $\mathcal{R}$ we have that $\sim_{\mathcal{R}}$ is also the convertibility relation over $\mathsf{T}_{\mathbf{A}} \times \mathsf{T}_{\mathbf{A}}$ generated by $\mathrm{Trs}^+(\mathcal{R})$. Now, given a s.r. $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ and a pair of types $A, B \in \mathsf{T}_{\mathbf{A}}$, we can easily decide whether $A \sim_{\mathcal{R}} B$ simply by checking that their (unique) normal forms with respect to $\mathrm{Trs}^+(\mathcal{R})$ are identical.

**Theorem 20.** *Let $\mathcal{R}$ be an expanded s.r.. Then $\sim_{\mathcal{R}}$ is decidable.* □

Another application of the properties of $\mathrm{Trs}^+(\mathcal{R})$ is also invertibility.

**Theorem 21.** *Let $\mathcal{R}$ be an expansion of a s.r.. Then $\sim_{\mathcal{R}}$ is invertible.*

*Proof.* Let $A \to B \sim_{\mathcal{R}} A' \to B'$. Let $X_N$ denote the normal form with respect to $\mathrm{Trs}^+(\mathcal{R})$ of type $X$ where $X$ is one of $A$, $B$, $A'$, $B'$. Then $X \sim_{\mathcal{R}} X_N$ for all $X$ and $A_N \to B_N \sim_{\mathcal{R}} A'_N \to B'_N$. Now if $A_N \to B_N$ is itself in normal form then, by uniqueness of normal forms, we must have $A_N \to B_N = A'_N \to B'_N$ and, then $A_N = A'_N$ and $B_N = B'_N$. By transitivity we have therefore $A \sim_{\mathcal{R}} A'$ and $B \sim_{\mathcal{R}} B'$.
Otherwise, if $A_N \to B_N$ is not in normal form, it must itself be a redex with

respect to $\mathrm{Trs}^+(\mathcal{R})$. Then $A_N \to B_N \rightsquigarrow c_i$ for some atomic type $c_i$ in a single step. Applying the same argument to $A'_N \to B'_N$ and by uniqueness of normal form we have also $A'_N \to B'_N \rightsquigarrow c_i$ and, since $\mathrm{Trs}^+(\mathcal{R})$ is invertible, this implies $A_N \to B_N = A'_N \to B'_N$. We conclude the proof as in the previous case. □

### 3.2 Other properties of weak equality

Some other properties of finitely presented type algebras will be useful in the study of typed $\lambda$-terms and, in particular, in the proof of decidability of type assignment.

It is sometimes useful to force a type algebra generated by a set of type constraints $\mathcal{C}$ to have the invertibility property. We write $\mathcal{C} \vdash^*_\sim A=B$ to mean that $A=B$ is provable in $(\sim)$ extended with the invertibility rule

$$(\mathrm{inv})\ \frac{\mathcal{C} \vdash A_1 \to A_2 = B_1 \to B_2}{A_i = B_i}\ (i = 1,2)$$

As in Definition 2 let $\sim^*_{\mathcal{C}}$ be the congruence defined this system. Note that $\sim^*_{\mathcal{C}}$ is the least invertible congruence containing $\sim_{\mathcal{C}}$. The construction presented in the following Lemma is due to Statman ([18])

**Lemma 22.** *Let $\mathcal{C}$ a system of type constraints. Then there is an expanded s.r. $\mathcal{C}^*$ such that $\mathcal{C} \vdash^*_\sim \mathcal{C}^*$ and $\mathcal{C}^* \vdash_\sim \mathcal{C}$ (i.e. $\mathcal{C}^*$ is equivalent to $\mathcal{C}$ plus invertibility).*

*Proof.* For all atomic types $c$ occurring in some equations of $\mathcal{C}$, let $[c]^*$ be the set $\{c' \mid c' \text{ is an atom and } \mathcal{C}^* \vdash^*_\sim c = c'\}$ (it is easy to design an algorithm to find all elements of $[c]^*$). For each equivalence class choose an atom $c^*$. Let now $\mathcal{C}^* = \mathcal{D}^* \cup \mathcal{E}^*$ where:

- $\mathcal{D}^*$ is the set of equations $c^* = A^*$, one for each class $[c]^*$, where $A^*$ is a non-atomic type expression of minimal length which contains only starred atoms and such that $\mathcal{C} \vdash^*_\sim c^* = A^*$.
- $\mathcal{E}^*$ is the set of all equations $a = c^*$ for all atomic types $a$ belonging to an equivalence class $[c^*]$.

It is obvious that $\mathcal{C} \vdash^*_\sim \mathcal{C}^*$.

We claim now that $\mathcal{C} \vdash^*_\sim A = B$ implies $\mathcal{C}^* \vdash_\sim A = B$. This implies that $\mathcal{C}^* \vdash_\sim \mathcal{C}$. We prove the claim by induction on the sum $\|A\| + \|B\|$. If both $A$ and $B$ are atomic types then the proof is trivial (they belong to the same equivalence class). Otherwise we can assume w. l. o. g. that both $A$ and $B$ contain only starred atoms. We have the following cases:
- If $A = A_1 \to A_2$ and $B = B_1 \to B_2$ we must have $\mathcal{C} \vdash^*_\sim A_i = B_i$ for $i = 1,2$, by invertibility, and the proof follows immediately from the induction hypothesis.
- If $A$ is an atom $a^*$ and $B = B_1 \to B_2$ then either $a^* = B \in \mathcal{D}^*$ and we are done or there is an equation $a^* = A_1 \to A_2 \in \mathcal{C}^*$ where both $A_i$ $(i = 1,2)$ contain only starred atoms and must be shorter or equal to $B_i$ (otherwise we could find a shorter non-atomic expression in $[a^*]^*$. Then $\mathcal{C} \vdash^*_\sim A_i = B_i$ for $i = 1,2$, by invertibility, and the proof follows easily from the induction hypothesis. □

Finally we need to introduce the notions of type algebra homomorphism and that of solavbility of a system of type equations in a s.r..

Let $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ and $\langle \mathsf{T}_{\mathbf{A}'}, \simeq' \rangle$ be type algebras. A mapping $h : \mathsf{T}_{\mathbf{A}} \to \mathsf{T}_{\mathbf{A}'}$ is a *type algebra homomorphism* if for all $A, B \in \mathsf{T}_{\mathbf{A}}$ such that $A \simeq B$ we have $h(A) \simeq' h(B)$.

We write $h : \langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle \to \langle \mathsf{T}_{\mathbf{A}'}, \simeq' \rangle$ to denote that $h$ is a type algebra homomorphism. It is easy to see that if $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ is invertible then $h$ is determined by its value on the atomic types of $A$.

**Definition 23.** Let $\mathcal{C}$ a system of type constrains over $\mathsf{T}_{\mathbf{A}}$. We say that a s.r. $\mathcal{R}$ over $\mathsf{T}_{\mathbf{A}'}$ *solves* $\mathcal{C}$ if there is a substitution $h : \mathbf{A} \to \mathsf{T}_{\mathbf{A}'}$ such that for all $A = B \in \mathcal{C}$ we have $h(A) \sim_{\mathcal{R}} h(B)$.

*Example 7.* Let $\mathcal{S} = \{a \to a = (a \to b) \to a\}$ be system of equations over $\mathsf{T}_{\{a,b\}}$. Then $\mathcal{S}$ has a solution in the s.r. $\{ c_1 = c_1 \to t\}$ via the substitution defined by $h(a) = c_1$, $h(b) = t$

The following result has been proved by R. Statman ([18]).

**Theorem 24.** *It is decidable whether a s.r. $\mathcal{R}$ solves a system $\mathcal{C}$ of equations over $\mathsf{T}_{\mathbf{A}}$.*

In [18] it is also shown that the solvability of a s.r. in another s.r. is indeed a NP-complete problem.

### 3.3 Axiomatization of strong equivalence

While the notion of weak equivalence was introduced by means of a formal inference system, that of strong equivalence was introduced in Chapter 2.1 in a semantic way (via the tree interpretation). We show in this section that also strong equivalence can be represented by a (rather simple) finite set of formal rules using a kind of coinduction principle.

A by-product of the proof of the completeness of these formalization is the decidability of strong equivalence.

The formal system $(\approx)$ for strong equivalence presented here is taken from Brandt and Henglein [3]. Other complete formalization of strong equivalence have been given by Amadio and Cardelli [1] and Ariola and Klop [2]. We will prove $(\approx)$ sound and complete for the infinite tree semantic introduced in Chapter 2.1. In this systems we have judgments of the form

$$\mathcal{R}, \mathcal{A} \vdash A = B$$

in which $\mathcal{A}$ represents set of equations of the shape $A \to A' = B \to B'$, where $A, B, A', B' \in \mathsf{T}_{\mathbf{A}}$. The meaning of this judgment is that we can prove $A = B$ from $\mathcal{R}$ assuming the equations in $\mathcal{A}$. We will show that provability in $(\approx)$ corresponds exactly to strong equivalence. In particular, when $\mathcal{A}$ is empty, this correspond to having $A \approx_{\mathcal{R}} B$.

This axiomatization contains a "co-inductive" rule of the form

$$\frac{\Gamma, P \vdash P}{\Gamma \vdash P}$$

provided that the proof of $P$ (where $P$ is a formula) has not been obtained in a trivial way from the assumption $P$ itself. This rule will be suitable to handle the infinitary nature of the tree semantics. It will indeed be given in a slightly different form (see the rule (coind) below) to guarantee that the restriction on the proof of $P$ has been met.

**Definition 25.** Let $\mathcal{R}$ denote a s.r over $\mathsf{T}_{\mathbf{A}}$. The system $(\approx)$ is defined by the rules (eq), (ident), (symm) and (trans) of Definition 2 plus the rules

$$\text{(hyp)} \quad \mathcal{R}, \mathcal{A} \vdash A = B \quad \text{if } A = B \in \mathcal{A}$$

$$\text{(coind)} \quad \frac{\mathcal{R}, \mathcal{A} \cup \{A \to B = A' \to B'\} \vdash_{\approx} A = A' \quad \mathcal{R}, \mathcal{A} \cup \{A \to B = A' \to B'\} \vdash_{\approx} B = B'}{\mathcal{R}, \mathcal{A} \vdash_{\approx} A \to B = A' \to B'}$$

We have not introduced in $(\approx)$ a rule for proving equality of arrow types of the shape

$$\frac{\mathcal{R}, \mathcal{A} \vdash_{\approx} A = A' \qquad \mathcal{R}, \mathcal{A} \vdash_{\approx} B = B'}{\mathcal{R}, \mathcal{A} \vdash_{\approx} A \to B = A' \to B'}$$

since this can be seen as a admissible rule in the system. In fact if $\mathcal{R}, \mathcal{A} \vdash_{\approx} A = A'$ then it is easy to see that $\mathcal{R}, \mathcal{A} \cup \{A \to B = A' \to B'\} \vdash_{\approx} A = A'$ (and similarly for $B = B'$).

*Example 8.* Let $\mathcal{R}'_1 = \{c = A \to A \to c\}$ where A is a any type be as in Example 1(ii). We have the following proof of $\mathcal{R}'_1 \vdash_{\approx} c = A \to c$. Let $T'$ denote $A \to A \to c$. We use $\{-\}$ as a shorthand for a set assumptions which it is not relevant in the statement in which occurs and we omit to mention $\mathcal{R}'_1$ in all the premises.

$$\dfrac{\dfrac{\text{(eq)}}{\vdash_{\approx} c = T'} \qquad \dfrac{\dfrac{\text{(ident)}}{\{-\} \vdash_{\approx} A = A} \qquad \dfrac{\dfrac{\text{(eq)}}{\{A \to c = T'\} \vdash_{\approx} c = T'} \qquad \dfrac{\text{(eq)}}{\{A \to c = T'\} \vdash_{\approx} A \to c = T'}\,\text{(trans)}}{\{A \to c = T'\} \vdash_{\approx} c = A \to c}\,\text{(coind)}}{\vdash_{\approx} T' = A \to c}\,\text{(trans)}}{\vdash_{\approx} c = A \to c}$$

To prove soundness and completeness of $(\approx)$ we need the notion of finite approximation of an infinite tree. Given $\alpha \in \mathsf{Tr}^{\infty}$, define for every $n \in \omega$ the tree $\alpha_{|n}$, its *truncation at level n*, as follows:

- $\alpha_{|0} = \Omega$, where $\Omega$ is a new constant symbol;
- $\kappa_{|n+1} = \kappa$, for $\kappa \in K \cup V$;
- $(\alpha' \to \alpha'')_{|n+1} = \alpha'_{|n} \to \alpha''_{|n}$.

Let $A, B \in \mathsf{T}_{\mathbf{A}}$. We write $A =_k^{\mathcal{R}} B$ if $(A^*_{\mathcal{R}})_{|k} = (B^*_{\mathcal{R}})_{|k}$.

We define now a semantic interpretation of the statements of $\vdash_{\approx}$. The definition is given by levels of approximations rather than directly.

**Definition 26.** Let $\mathcal{R}$ be a system of type constraints.

(i) $\mathcal{R} \models_k A = B$ if $A =_k^{\mathcal{R}} B$.
(ii) $\mathcal{R} \models_k \mathcal{A}$ if for all $A = B \in \mathcal{A}$ $\mathcal{R} \models_k A = B$.
(iii) $\mathcal{R}, \mathcal{A} \models_k A = B$ if $\mathcal{R} \models_k \mathcal{A}$ implies $\mathcal{R} \models_k A = B$.
(iv) $\mathcal{R}, \mathcal{A} \models A = B$ if for all $k \geq 0$ $\mathcal{R}, \mathcal{A} \models_k A = B$

We note, in particular, that $\mathcal{R} \models A = B$ implies that $A^*_{\mathcal{R}} = B^*_{\mathcal{R}}$, i.e. $A \approx_{\mathcal{R}} B$.

**Theorem 27.** *(Soundness)* $\mathcal{R}, \mathcal{A} \vdash_{\approx} A = B$ *imply* $\mathcal{R}, \mathcal{A} \models A = B$

*Proof.* the proof is by induction on derivations. For the axioms (ident) and (eq) and for rule (trans) the proof is trivial. As for rule (coind) we prove that if, $\mathcal{R}, \mathcal{A} \vdash_{\approx} A \to B = A' \to B'$ ha been obtained by (coind), for all $k \geq 0$ $\mathcal{R}, \mathcal{A} \models_k A \to B = A' \to B'$. This is done by induction on $k$. The case $k = 0$ is trivial since $A =_k^{\mathcal{R}} B$ for all types $A, B$. Let now $k \geq 0$, and assume $\mathcal{R} \models_k \mathcal{A}$. Then also $\mathcal{R} \models_{k-1} k\mathcal{A}$ and this, by induction hypothesis on $k$ implies $A \to B =_{k-1}^{\mathcal{R}} A \to B'$. Then $\mathcal{R} \models_{k-1} \mathcal{A} \cup \{A \to B = A' \to B'\}$. By induction hypothesis on derivations this implies $A =_{k-1}^{\mathcal{R}} A'$ and $B =_{k-1}^{\mathcal{R}} B'$ and, then, $A \to B =_k^{\mathcal{R}} A \to B'$. This concludes the proof.

When $\mathcal{A}$ is empty, we get the soundness of $(\approx)$ with respect to $\approx_\mu$.

The proof of completeness of $\vdash_{\approx}$ is given in a constructive way. In the following definition, given a s.r. $\mathcal{R}$ and a type equality $A = B$ we define a process that either fails or gives a proof of $A = B$ in $\mathcal{R}$. The proof is built by defining a sequence $\mathcal{S}_m$ $(m \geq 0)$ of sets of pairs of the shape $\langle \mathcal{A}, A' = B' \rangle$ (plus a distinguished element FAIL) such that, for each $m$, $\mathcal{R} \vdash_{\approx} A = B$ con be proved in $\vdash_{\approx}$ assuming $\mathcal{R}, \mathcal{A} \vdash_{\approx} A' = B'$ for each pair $\langle \mathcal{A}, A' = B' \rangle \in \mathcal{S}_m$. We will prove that, if $\mathcal{R} \models A = B$, we will reach an $m$ for which $\mathcal{S}_m$ is empty and so that $\mathcal{R} \vdash_{\approx} A = B$ is provable.

**Definition 28.** Let $\mathcal{R} = \{c_i = C_i \mid 1 \leq i \leq n\}$ be a s.r. and $A = B$ a type equations. We define a succession $\mathcal{S}_m$ $(m \geq 0)$ where $\mathcal{S}_m$ is either a set of pairs of the shape $\langle \mathcal{A}, A = B \rangle$ where $A = B$ is a type equation and $\mathcal{A}$ is a set of type equations or a distinguished element FAIL.
Let $\mathcal{S}_0 = \{\langle \emptyset, A = B \rangle\}$.
For $m > 0$ define $\mathcal{S}_m$ from $\mathcal{S}_{m-1}$ in the following way. Take any pair $\langle \mathcal{A}, A = B \rangle \in \mathcal{S}_{m-1}$ and let $\mathcal{S}' = \mathcal{S}_{m-1} - \{\langle \mathcal{A}, A = B \rangle\}$.
Let $A' = A$ if $A$ is not an unknown $c_i$ of $\mathcal{R}$ and $A' = C_i$ if $A$ is the unknown $c_i$ of $\mathcal{R}$. Similarly let $B' = B$ if $B$ is not an unknown $c_j$ of $\mathcal{R}$ and $B' = C_j$ if $B$ is the unknown $c_j$ of $\mathcal{R}$ Then we have the following cases

**Case 1** If $A' = B'$ or $A' = B' \in \mathcal{A}$ then $\mathcal{S}_m = \mathcal{S}'$.

**Case 2** If $A'$ and $B'$ are different constants or variables, or one is a constant or variable and the other is an arrow type then $\mathcal{S}_m = \text{FAIL}$.

**Case 3** Otherwise $A' = A_1 \to A_2$ and $B' = B_1 \to B_2$.
Now let $I = \{i \mid A_i = B_i \notin \mathcal{A} \text{ and } A_i \neq B_i\}$ (obviously $I$ has 0, 1 or 2 elements). Then define

$$\mathcal{S}_m = \mathcal{S}_{m-1} \cup \{\langle \mathcal{A} \cup \{A_1 \to A_2 = B_1 \to B_2\}, A_i = B_i\rangle \mid i \in I\}$$

**Lemma 29.** *The sequence $\mathcal{S}_m$ defined in Definition 28 is finite, i.e. for some finite $n$ either $\mathcal{S}_m = FAIL$ or $\mathcal{S}_m$ is empty.*

*Proof.* It is easy to see, by induction on $m$, that each each pair $\langle \mathcal{A}, A' = B'\rangle$ that is put i some $\mathcal{S}_m$ is such that $A', B'$ and all the types occurring in $\mathcal{A}$ must are subtypes of $A, B$ or of some type $C_i$ for $(i \leq i \leq n)$. So there is only a finite number $K$ of possible equations $A' = B'$ that can occur in the pairs of $\mathcal{S}_m$.
Now define the *length* of a pair $\langle \mathcal{A}, A' = B'\rangle$ as the number of equations in $\mathcal{A}$ and note that either $\mathcal{S}_m$ has one pair less that $\mathcal{S}_{m-1}$ (in Case 1, 2 and sometimes in Case 3) or the new pairs added in $\mathcal{S}_m$, in Case 3, have length greater than the pair that has been eliminated. But no pair of length $K+1$ can be put in any $\mathcal{S}_m$ since in this case, in Case 3, $A_i = B_i$ must belong to $\mathcal{A}$ for $i = 1, 2$ since $\mathcal{A}$ contains all possible equations. Then if FAIL does not occur in the sequence of the $\mathcal{S}_m$ we must eventually reach a point $m_0$ for which $\mathcal{S}_{m_0} = \emptyset$.

The following properties are easily proved by induction.

**Lemma 30.** *Let $\mathcal{R} \models A = B$ and $\mathcal{S}_m$ be defined as in Def. 28.*

**(i)** *For all pairs $\langle \mathcal{A}, A' = B' \in \mathcal{S}_m\rangle$ we have that $\mathcal{R}, \mathcal{A} \models A' = B'$.*
**(ii)** *The sequence of the $\mathcal{S}_m$ cannot end with FAIL.*

**Lemma 31.** *Let $\mathcal{S}_m$ be defined as in Def. 28 ($n \geq 0$). If we assume to have a proof of $\mathcal{R}, \mathcal{A} \vdash_{\approx} A' = B'$ for each pair $\langle \mathcal{A}, A' = B'\rangle \in \mathcal{S}_m$ then we can build a proof of $\mathcal{R} \vdash_{\approx} A = B$.*

The completeness Theorem follows immediately.

**Theorem 32.** $\mathcal{R} \vdash_{\approx} A = A$ *iff* $\mathcal{R} \models A = B$.

## 4 Properties of typed terms

In this section we establish some basic properties of terms which have types in the previous systems.

### 4.1 Subject reduction

We show now that recursive type system have essentially the subject reduction property, i.e. if $\mathcal{R}, \Gamma \vdash M : A$ where $\vdash$ is one of the systems defined in the previous chapter and $M \longrightarrow_\beta M'$ (or $M \longrightarrow_{\beta\eta} M'$ then also $\Gamma \vdash M' : A$. This property is important for type systems since it means that typings are stable with respect to reduction, which is the fundamental evaluation process for $\lambda$-terms. Types are not preserved, in general, by the reverse operation of expansion. We will study subject reduction for systems induced by a generic type algebra $\langle \mathsf{T}_\mathbf{A}, \simeq \rangle$ and we will indicate by $\vdash_{\lambda\simeq}$ provability in the system which uses $\simeq$ as type equivalence. The main result of this section, which has been first proved by Statman [18] is that $\vdash_{\lambda\simeq}$ as the subject reduction property iff $\langle \mathsf{T}_\mathbf{A}, \simeq \rangle$ is invertible. Our treatment is based essentially on the proof of Marz [12].

**Lemma 33.** *(i) If $\Gamma \vdash_{\lambda\simeq} x : A$ where $x$ is a variable then $x : A' \in \Gamma$ for some type $A'$ such that $A' \simeq A$.*
*(ii) If $\Gamma \vdash_{\lambda\simeq} \lambda x.M : A$ then there are types $B$ and $C$ such that $A \simeq (B \to C)$ and $\Gamma \cup \{x : B\} \vdash_{\lambda\simeq} M : C$.*
*(iii) If $\Gamma \vdash_{\lambda\simeq} (MN) : A$ then there exists a type $B$ such that $\Gamma \vdash_{\lambda\simeq} M : B \to A$ and $\Gamma \vdash_{\lambda\simeq} N : B$.*

*Proof.* (i) To prove $\Gamma \vdash_{\lambda\simeq} x : A$ we can use only rule (ax) and (equiv). Now note that all relations $\simeq$ are transitive.
(ii) The proof of $\Gamma \vdash_{\lambda\simeq} \lambda x.M : A$ must end with an application of ($\to$-intro) followed, possibly, by a number of applications of (equiv). Let $\Gamma \cup \{x : B\} \vdash_{\lambda\simeq} M : C$ be the premise of the application of ($\to$-intro) and $\Gamma \vdash_{\lambda\simeq} \lambda x.M : B \to C$ its conclusion. Again, since $\simeq$ is transitive, we have $(B \to C) \simeq A$.
(iii) Arguing as before we conclude that there in the proof of $\Gamma \vdash_{\lambda\simeq} (MN) : A$ there is an application of rule ($\to$-intro) whose premises are $\Gamma \vdash_{\lambda\simeq} M : B \to A'$ and $\Gamma \vdash_{\lambda\simeq} N : B$ where $A' \simeq A$. We have also $(B \to A') \simeq (B \to A)$ and so we can prove $\Gamma \vdash_{\lambda\simeq} M : B \to A$ from $\Gamma \vdash_{\lambda\simeq} M : B \to A'$.

We can now state the basic lemmas needed for the proof of the subject reduction property. Note that invertibility plays an essential role here.

**Lemma 34.** *Let $\langle \mathsf{T}_\mathbf{A}, \simeq \rangle$ be an invertible type algebra.*

*(i) If $\Gamma \vdash_{\lambda\simeq} (\lambda x.M)N : A$ then $\Gamma \vdash_{\lambda\simeq} (M[x := N]) : A$.*
*(ii) If $\Gamma \vdash_{\lambda\simeq} \lambda x.(Mx) : A$, where $x$ does not occur in $M$, then $\Gamma \vdash_{\lambda\simeq} M : A$.*

*Proof.* (i) By the Lemma 33(iii) we have that
$\Gamma \vdash_{\lambda\simeq} (\lambda x.M) : B \to A$ and $\Gamma \vdash_{\lambda\simeq} N : B$ for some types $B$. Moreover, by Lemma 33(ii), we have that for some types $B'$ and $A'$

$$(1)\ \Gamma \cup \{x : B'\} \vdash_{\lambda\simeq} M : A'$$

where $B' \to A' \simeq B \to A$. Since $\simeq$ is invertible we have also $B' \simeq B$ and $A' \simeq A$. We can then obtain a deduction of

$$(2)\ \Gamma \vdash_{\lambda\simeq} (M[x := N]) : A'$$

by replacing, in the deduction of (1), each use of the premise $x : B'$ with a copy of the deductions of $\Gamma \vdash_{\lambda\simeq} N : B$ followed by an application of rule (equiv) (if necessary) using $B' \simeq B$. Eventually we can get a deduction of $\Gamma \vdash_{\lambda\simeq} (M[x := N]) : A$ from (2) by applying rule (equiv) (if necessary) using $A' \simeq A$.
(ii) By Lemma 33(ii) we have that $\Gamma \cup \{x : B\} \vdash_{\lambda\simeq} (Mx) : C$ for some types $B, C$ such that $A \simeq (B \to C)$. Moreover, by Lemma 33 (iii) and (i) we have that $\Gamma \cup \{x : B\} \vdash_{\lambda\simeq} M : D \to C$ and $\Gamma \cup \{x : B\} \vdash_{\lambda\simeq} x : D$ for some type $D$ such that $D \simeq B$. Since $\simeq$ is a congruence, we have $(D \to C) \simeq (B \to C) \simeq A$ and then $\Gamma \vdash_{\lambda\simeq} M : A$ using (possibly) (equiv) and observing that $x$ does not occur in $M$. □

**Theorem 35.** *(Subject Reduction) Let $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ be an invertible type algebra. Suppose $\Gamma \vdash_{\lambda\simeq} M : A$ and $M \longrightarrow_{\beta\eta} M'$. Then $\Gamma \vdash_{\lambda\simeq} M' : A$*

*Proof.* By induction on the generation of $\longrightarrow_\beta$ using Lemma 34. □

In particular $(\approx)$ and $(\sim)$ when weak equality is induced by a s.r. (see Theorem 21) have the subject reduction property.

An important consequence of the subject reduction theorem, especially from the computational point of view, is the fact that in evaluating a well typed term using $\beta$-reduction all redexes are always well-typed, i.e. when evaluating a typeable term we will never reach an application in which the term in function position cannot be used as a function. So for instance it cannot happen that an integer is applied as a function to an argument. The same property also holds in systems with constants, if these are given the proper type. So a well typed program will never produce a run-time error caused by an incorrect application. This property is, from the point of view of computer science, even more important that the strong normalization property. In fact all "real" functional programming languages have fixpoint operators at all types which allow to define non terminating computations.

In particular it can be proved that only invertible type algebras induce a type inference system with the subject reduction property. We say that a type algebra $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ *satisfies* the subject reduction Property if, whenever $\Gamma \vdash_{\lambda\simeq} M : A$ and $M \longrightarrow_\beta M'$ (or $M \longrightarrow_{\beta\eta} M'$), then also $\Gamma \vdash_{\lambda\simeq} M' : A$.

**Theorem 36.** *A type algebra $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ satisfies the subject reduction Property only if is invertible*

For the proof see Marz ([12], Proposition 1.16).

## 4.2 Finding types

In this section we consider the problem of deciding whether a given term has a type in an inference system with recursive types. There are several questions that can be asked about the typeability of $M$. In particular we will address the following questions, formulated here for weak equality. The same questions can of course be asked also for strong equality.

Assume that $M$ is a closed term.

1. Does it exists a s.r. $\mathcal{R}$ and a type $A$ such that $\mathcal{R} \vdash_{\lambda\sim} M : A$ ?
2. Given a s.r. $\mathcal{R}$, does there exist a type $A$ such that $\mathcal{R} \vdash_{\lambda\sim} M : A$

Note that, from Lemma 33 $\{x_1 : A_1, \ldots, x_n : A_n\} \vdash_{\lambda\simeq} M : A$ iff $\vdash_{\lambda\simeq} \lambda x_1 \ldots x_n.M : A_1 \to \ldots \to A_n \to A$. So it is not restrictive to ask these questions for a closed term only.

If $M$ is a pure $\lambda$-term (without constants) question 1. is trivially decidable since all pure $\lambda$ terms have a type in $\vdash_{\lambda\simeq}$ assuming a type $c$ such that $c = c \to c$ (see Example 1 (i)). This is however not true anymore if $M$ contains constants.

We will show in this section that all these questions, for both weak and strong equivalence, are decidable. In all cases the basic tool to show this is a generalization of the notion of principal type scheme.

To make the treatment of this section more straightforward we will prove the main properties for pure $\lambda$-terms (without constants). The extension of these properties to terms with constants will be discussed in Remark 4.2.
In the following definition we assume, without loss of generality, that all free and bound variables in a term have distinct names.

**Definition 37.** Let $M$ be a term. The system of type constraints $\mathcal{C}_M$, the basis $\Gamma_M$ and type $T_M$ are defined as follows. Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1$ is set of all bound and free variables of $M$ and $\mathcal{S}_2$ is the set of all occurrences of subterms of $M$ which are not variables. Take now a set $\mathbf{I}_M$ of atomic type in bijective correspondence with $\mathcal{S}$ and assign a different type $i \in \mathbf{I}_M$ to each element $P$ of $\mathcal{S}$. Let $\pi(P)$ denote the atomic type assigned to $P$. Then define $\mathcal{C}_M$ over $\mathsf{T}_{\mathbf{I}_M}$ by adding an equation for each element $P$ of $\mathcal{S}_2$ in the following way.

(a) If $P = \lambda x.P_1$ let $i_1 = \pi(\lambda x.P_1), i_2 = \pi(x), i_3 = \pi(P_1)$. Then put $(i_1 = i_2 \to i_3)$ in $\mathcal{C}_M$ .
(b) If $P = (P_1 P_2)$ let $\pi(P_1) = i_1, \pi(P_2) = i_2$ and $\pi(P_1 P_2) = i_3$ then put $(i_1 = i_2 \to i_3)$ in $\mathcal{C}_M$.

Moreover let $\Gamma_M = \{x : \pi(x) \mid X \text{ is free in } M\}$ and $T_M = \pi(M)$.

It is immediate to prove, by induction on $M$, that $\mathcal{C}_M, \Gamma_M \vdash_{\lambda\sim} M : T_M$.

Note that $\mathcal{C}_M$ is a system of type constraints but not, in general, a s.r.. In fact in general we can have many equations with the same left hand side in $\mathcal{C}_M$. Now, assuming invertibility and applying Lemma 22, we can transform $\mathcal{C}_M$ into an expanded s.r. $\mathcal{C}^*_M$. As remarked in Chapter 3, $\sim_{\mathcal{C}^*_M} = \sim^*_{\mathcal{C}_M}$ is the least invertible type congruence extending $\sim_{\mathcal{C}_M}$.

Since $\sim_{\mathcal{C}^*_M}$ extends $\sim_{\mathcal{C}_M}$ it is obvious that also $\mathcal{C}^*_M, \Gamma_M \vdash_{\lambda\sim} M : T_M$.

*Example 9.* Consider the term $\lambda x.xx$ and take $\pi(x) = i_1$, $\pi(xx) = i_2$, $\pi(\lambda x.xx) = i_3$. Then we have

$$\mathcal{C}_{\lambda x.xx} = \{i_1 = i_1 \to i_2,\ i_3 = i_1 \to i_2\}$$

Applying to this system the construction in Lemma 22 we get $\mathcal{D}^* = \{i_1 = i_1 \to i_2\}$ and $\mathcal{E}^* = \{i_3 = i_1\}$, where we have taken $i_1$ as the representative of the equivalence class $[i_1]$ containing $i_3$. So we have $\mathcal{C}^*_{\lambda x.xx} = \{i_1 = i_1 \to i_2, i_3 = i_1\}$, $\Gamma^*_{\lambda x.xx} = \emptyset$, $T_{\lambda x.xx} = i_3$.

**Theorem 38.** *Let $M \in \Lambda$ and $\langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ be an invertible type algebra such that $\Gamma \vdash_{\lambda\simeq} M : A$ for some type $A$ and basis $\Gamma$. Then there is a type algebra homomorphism $\phi : \langle \mathsf{T}_{\mathbf{I}_M}, \sim_{\mathcal{C}^*_M} \rangle \to \langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ such that $A = \phi(T_M)$ and $\Gamma = \phi(\Gamma_M)$.*

*Proof.* Take a deduction D of $\Gamma \vdash_{\lambda\simeq} M : A$. For $P \in \mathcal{S}$ (see Def. 37) let $\tau(P)$ be the type assigned to $P$ in D. Note that, by Lemma 33 $\tau(P)$ is well defined, modulo $\simeq$. Now we have seen in Chapter 3 that an homomorphism between invertible type algebras is determined by its values on atomic types. So let $\phi : \langle \mathsf{T}_{\mathbf{I}_M}, \sim_{\mathcal{C}^*_M} \rangle \to \langle \mathsf{T}_{\mathbf{A}}, \simeq \rangle$ be the homomorphism defined by

$$\phi(i) = B \text{ if, for some } P \in \mathcal{S},\ \pi(P) = i \text{ and } \tau(P) = B.$$

Obviously $\Gamma = \phi(\Gamma_M)$ and $A = \phi(\pi(M))$. We now prove that $\phi$ is a type theory isomorphism. We have only to prove that $A \sim_{\mathcal{C}^*_M} B$ implies $\phi(A) \simeq \phi(B)$. Since $\mathcal{C}^*_M$ is logically equivalent to $\mathcal{C}_M$ plus invertibility it is enough to prove that for all equations $(i_1 = i_2 \to i_3) \in \mathcal{C}_M$ $\phi(i_1) \simeq \phi(i_2) \to \phi(i_3)$. We distinguish two cases according to the equation has been put in $\mathcal{M}$ by case (a) or (b) above.
In case (a) let $A_1 = \tau(\lambda x.P_1)$, $A_2 = \tau(x)$ and $A_3 = \tau(P_1)$. We have by definition $A_k = \phi(i_k)$ $(k = 1, 2, 3)$. By Lemma 33(ii) we must have $A_1 \simeq A_2 \to A_3$.
The case (b) is handled similarly, using Lemma 33(iii). □

$\mathcal{C}^*_M$ can be simplified in two ways. All atomic types $i_1, i_2$ such that $i_1 = i_2 \in \mathcal{C}^*_M$ can be identified and replaced with a single atomic type. Moreover, any equation $i = C \in \mathcal{C}^*_M$ where the variable $i$ does not occur in $C$ can be eliminated simply by replacing $i$ with $C$ in both $\Gamma_M$, $A_M$ and $\mathcal{C}^*_M$. It is easy to define an algorithm to perform such simplifications. Let $\overline{\mathcal{R}}_M$, $\overline{\Gamma}_M$ and $\overline{T}_M$ be the results of these simplifications. It is immediate that $\overline{\mathcal{R}}_M, \overline{\Gamma}_M \vdash_{\lambda\sim} \overline{T}_M$ and that Theorem 38 holds for $\overline{\mathcal{R}}_M$, $\overline{\Gamma}_M$ and $\overline{T}_M$ as well.

$\overline{\mathcal{R}}_M$, $\overline{\Gamma}_M$ and $\overline{T}_M$ are the generalization of the notion of principal type and basis scheme for simple types (see Hindley [8, 9]). Note that the equations in $\overline{\mathcal{R}}_M$ represent the weakest recursive definitions needed to give a type to $M$, and this for both the weak and strong equivalence. So a term is typeable in the weak system if and only if it is typeable in the strong one. It is easy to see that if a term is typeable in the Curry system we get an empty $\overline{\mathcal{R}}_M$.

We can now give a classification of the atomic types of $\mathbf{I}_M$ of Definition 37. The elements of $\mathbf{I}_M$ which do not occur in the left hand side of an equation of $\overline{\mathcal{R}}_M$ can indeed be mapped into any type via a type algebra homomorphism. They can then be considered as type variables, as in the construction of the principal type schemes in Curry's system. On the contrary, the atomic types

$i$ which occur in the left hand side of an equation of $i = A \in \overline{\mathcal{R}}$ cannot be mapped into arbitrary types by a type algebra homomorphism $\phi$ since $\phi(i)$ must be equivalent to $\phi(A)$ in the target type algebra. It is more natural to consider them as constants, which represent the recursive types needed to type the given term.

*Example 10.* Let us continue Example 9. We can simplify $\mathcal{C}_{\lambda x.xx}$ by identifying $i_1$ and $i_3$ and obtain the principal typing: $\{i_1 = i_1 \to t\} \vdash_{\lambda\sim} \lambda x.xx : i_1 \to t$ where $t$ is indeed a type variable.

*Remark.* If $M$ is not a pure $\lambda$ term then we can build $\mathcal{C}_M$ as in Definition 37 considering a constant $c$ as a free variables but we must add to $\mathcal{C}_M$ also the equations $i = T_c$ where $i = \phi(c)$ and $T_c$ is the type associated to the constant $c$. Then $M$ has a type only if $\mathcal{C}_M$ is *consistent* in the sense that $\mathcal{C}_M \not\vdash_\sim \kappa = \chi$ where $\kappa$ is a type constant and $\chi$ is either a different type constant or a $\to$-type. It is easy to check the consistency of a system of equations.
In considering the notion of type algebra homomorphism, moreover, we must add the condition that type constants are mapped into themselves. In fact the type of the (term) constants is independent of recursive type definitions.

Note that, while for each pure term $M$ there is at last one s.r. from which $M$ can be typed, this is not true for terms with constants. Take for instance the term $(\lambda x.x\ 1)2$. Indeed, by the subject reduction property, the evaluation of a term typeable in any consistent set of type constrains do not generate bad applications like $(2\ 1)$ as in the above case. In this sense the consistency of $\mathcal{C}_M$ is enough to guarantee the good evaluation of $M$.

Let now $\mathcal{R}$ be a given s.r.. By Theorem 38, given a term $M$, we can give a type to $M$ from $\mathcal{R}$ only if we find an homomorphism $h$ form $\langle \mathsf{T}_{\mathbf{I}_M}, \sim_{\overline{\mathcal{R}}_M} \rangle$ to $\langle \mathsf{T}_{\mathbf{A}}, \sim_{\mathcal{R}} \rangle$. By Theorem 21 this amounts to solve $\overline{\mathcal{R}}_M$ in $\mathcal{R}$. Since this property is decidable (Lemma 24) we have the following consequence.

**Theorem 39.** *Given a term $M$ and a s.r. $\mathcal{R}$ it is decidable whether $M$ is typeable form $\mathcal{R}$ in $(\lambda\sim)$, i.e. whether there is a basis $\Gamma$ and a type $A$ such that $\mathcal{R}, \Gamma \vdash_{\lambda\sim} M : A$.*

We cannot extend immediately to strong equivalence the proof of the previous theorem In fact the proof of solvability of a s.r. in another is based on a notion of term rewriting system which is not immediately transferable to strong equivalence. We left as open the problem of proving that typeability with respect to a given s.r. is decidable also for strong equivalence.

### 4.3 About Strong normalization

We have seen from the examples in Section 2.1 that the systems with recursive types do not have, in general, the strong normalization property. However it is possible to define a class of s.r. (the *inductive* ones) that guarantees that all term

typed from them with weak equality are strongly normalizable. The condition of being inductive gives indeed a characterization of all s.r. which guarantees strong normalization. We conjecture that a similar characterization could be given also for strong equality, but at the author's knowledge no result of this kind has never been published.

The result in this section are due essentially to Mendler [14], [13].

Let $A \in \mathsf{T}_{\mathbf{A}}$ and $B$ be an occurrence of a subtype of $A$. We say that $B$ is *positive* (*negative*) in $A$ if $B$ occurs in $A$ on the left hand side of an even (odd) number of '$\rightarrow$'. Let this number be the *level* of the occurrence of $B$ in $A$.

In [14] and [13] Mendler has characterized a class of s.r.such that all terms typeable from them are strongly normalizable.

**Definition 40.** A s.r. $\mathcal{R}$ is *inductive* if for no atomic type $c$ we have $c \sim_{\mathcal{R}} C$ for some non atomic type expression $C$ in which $c$ has a negative occurrence.

**Theorem 41.** *Let $\mathcal{R}$ be an inductive s.r.. Then $\Gamma, \mathcal{R} \vdash_{\lambda\sim} M : A$ implies that $M$ is strongly normalizing.*

The proof is given in [14], [13]. In the same papers It is proved also that the condition of being inductive characterizes exactly all the s.r which guarantee strong normalization.

**Theorem 42.** *Let $\mathcal{R}$ be a non inductive s.r.. Then there is a term $N$ without normal form such that for some basis $\Gamma$ we have $\mathcal{R}, \Gamma \vdash_{\lambda\sim} N : c$ where $c$ is an atomic type such that $c \sim_{\mathcal{R}} C$ for some non atomic type expression $C$ in which $c$ occurs negatively.*

It can be easily proved that, given a s.r. $\mathcal{R}$, it is decidable whether $\mathcal{R}$ is inductive of not.

One natural question to ask is whether a term $M$ is typeable also with respect to inductive s.r.. The following theorem ([12]) answer it.

**Theorem 43.** *Given a term $M$ it is decidable whether there is an inductive s.r. $\mathcal{R}$ such that we can give a type to $M$ from $\mathcal{R}$, i.e. such that $\mathcal{R}, \Gamma \vdash_{\lambda\sim} M : A$ for some type $A$ and basis $\Gamma$.*

*Proof.* Take the s.r. $\mathcal{C}^*_M$. If $\mathcal{C}^*_M$ is inductive then we can give a positive answer to the problem.

Otherwise assume that $\mathcal{C}^*_M$ is not inductive and take any s.r. $\mathcal{R}$ over $\mathsf{T}_{\mathbf{A}}$ such that a type can be given to $M$ from $\mathcal{R}$. Then by Theorem 38 there is type algebra homomorphism

$$\phi : \langle \mathsf{T}_{\mathbf{I}_M}, \sim_{\mathcal{C}^*_M} \rangle \rightarrow \langle \mathsf{T}_{\mathbf{A}}, \sim_{\mathcal{R}} \rangle$$

Now take $i \in \mathbf{I}_M$ such that $i \sim_{\mathcal{C}^*_M} C$ such that $i$ occurs negatively in $C$. Then we must have $\phi(i) \sim_{\mathcal{R}} \phi(C)$. Now it is easy to prove, by induction on $\phi(i)$ and exploiting the fact that $\sim_{\mathcal{R}}$ is invertible, that this implies that there is an atomic type $d \in \mathbf{A}$ such that $d \sim_{\mathcal{R}} D$ where $d$ occurs negatively in $D$.

This proves that a term $M$ can be typed from an inductive s.r. if and only if $\mathcal{C}^*_M$ is inductive. This last fact is obviously decidable. □

*Example 11.* Take the term $\lambda x.xx$. We have seen in Example 9 that $\mathcal{C}_{\lambda x.xx}$ contains an equation $i_1 = i_1 \to i_2$ and then is not inductive. Then there is no inductive s.r. which can give a type to $\lambda x.xx$.

## References


1. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
2. Z.M. Ariola and J.W. Klop. Equational term graph rewriting. Technical report, University of Oregon, 1995.
3. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In P. de Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer-Verlag, 1997.
4. V. Breazu-Tannen and A. Meyer. Lambda calculus with constrained types. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 1985.
5. F. Cardone and M. Coppo. Type inference with recursive types. Syntax and Semantics. *Information and Computation*, 92(1):48–80, 1991.
6. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
7. H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.
8. J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
9. J.R. Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1997.
10. H.P.Barendregt. Labda calculi with types. In Dov M. Gabbay S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages ??–?? Oxford University Uress, New York, 1992.
11. J.W. Klop. Term rewriting systems. In Dov M. Gabbay S. Abramsky and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Uress, New York, 1992.
12. M. Marz. An algebraic view on recursive types. Preprint 1793, Technische Hochschule Darmstadt, 1995. Revised version, May 29, 1996.
13. N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51:159–172, 1991.
14. P.F. Mendler. Inductive definitions in type theory. Technical Report 87-870, Department of Computer Science, Cornell University, Ithaca, New York, 1987. Ph. D. Thesis.
15. R. Milner. A Theory of Type Polimorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
17. D. Scott. Some philosophical issues concerning theories of combinators. In C. Böhm, editor, *Lambda calculus and computer science theory*, volume 37 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 1975.
18. R. Statman. Recursive types and the subject reduction theorem. Technical Report 94–164, Carnegie Mellon University, 1994.

19. D.A. Turner. An overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison Wesley, 1990.

This article was processed using the LaTeX macro package with LLNCS style